Exploring Machine Learning to assign debug priorities to improve the design quality

Vyasa Sai, Vaibhav Gupta and Fylur Rahman Sathakathulla
Intel Corporation
1900 Prarie City Road
Folsom, CA 95630

***Abstract*-Verification has become more challenging with the rapid increase in design complexity. Typically, regression test failures are bucketized, and then debug is prioritized for those tests which have similar failure signatures. Most efforts in verification have been used to bucketize test failures and not on identifying the critical failures with higher debug priority. But these prioritized failures may or may not end up as a design bug, thereby impacting the time to market. Hence, there is a need to explore new approaches applicable to the debug process that would accelerate finding of crucial design bugs. Machine Learning (ML) is emerging as a promising tool in the field of verification. In this paper, a methodology is proposed that applies ML techniques to analyze past failures to detect if a current regression failure is a design issue or not. As part of the proposed solution, an ML model was trained with past failure records and tested with current failures to be able to predict the failure as a design bug or not. This exploratory work introduces a novel methodology that integrates ML with a debug automation framework to accelerate design bug hunting. Different ML models were explored for this ML problem with the best model having a design recall of 0.82 for the given dataset. In other words, the proposed solution of applying ML to the verification domain has shown promise for design bug predictions. Finally, the paper also discusses the key challenges and outlines future directions for applications of the proposed research.**

I. Introduction

Typically, the design verification flow exercises critical test scenarios in the design through numerous simulations. One of the well-known verification processes is the use of regressions that enables running numerous test cases to hit different verification scenarios that usually take longer but provide a higher probability of finding bugs. As hardware designs grow in size and complexity, verification becomes increasingly challenging. Verification today takes up to more than 50% of the time spent in all design development cycles [1], [2], [3].

In most design verification flows, regression test failures are triaged and then prioritized based on maximum repetition of a similar failure signature. Automation as part of the verification process has mostly been used to triage test failures and not on identifying the failures. But these prioritized debugs may or may not end up as a design bug. Hence crucial debug time may be spent on non-design issues that could end up pushing out design delivery deadlines in turn impacting the time to market. Thus, there is a need to develop verification methodologies that are efficient, faster and less expensive.

Recently there has been an increase in applying Machine Learning (ML) to the field of verification both across the industry and research community [3], [4], [5], [6], [7], [8]. Any bug in hardware compared to software or firmware is a higher business risk as it is more expensive and time-consuming to fix. To help expedite successful hardware design tape-out schedules under time to market pressure, novel optimizations in design verification play a significant role in this context. Hence, there is a need to explore ML approaches which can be applied to the debug process that would help accelerate design bug hunting. Even though there has been ML techniques used to predict software issues [4], but application of ML to hardware related issues has been relatively new [5],[8]. Some of the main applications of ML to verification includes approaches such as guiding the input stimulus, predicting RTL bugs using design code changes, etc. [5], [6], [7], [8].

In this paper, a pre-silicon verification methodology is proposed to accelerate the design bug finding in regressions on IP level designs using an ML assisted debug automation framework. The impact of such a methodology enables faster time to market, quality IP delivery with enhanced debug throughput, and reduced engineering effort. This methodology can be applied to smaller (e.g., block or subsystem) as well as larger designs (SoC etc.).

The rest of the paper is organized as follows. Section II elaborates the proposed solution with details on the dataset (training and test) description, data wrangling criteria, ML model exploration and integration into a debug automation framework. Section III discusses the performance of the methodology along with details on preliminary

results of the best ML model and challenges faced. Finally, Section IV concludes the paper with insights for future work.

II. METHODOLOGY

A. *Proposed solution*

One of the objectives of the proposed solution is to identify whether a particular failure in a regression indicates a possible RTL design issue based on the historical data captured for previous failures using ML. The first step in the ML part of the proposed solution is to formulate the problem based on the data available and the desired outcome. The second step involves selecting inputs and outputs of the ML model. The captured error message(s) appearing in the regression logs will be the input to the ML model. This ML model classifies an input message as a possible design or non-design (testbench, BFM etc.) issue based on the learning from past failures. The ML model inputs and outputs are denoted as *error* and *bug-type* respectively. Thus, the ML problem maps to a binary classification problem with text data as input. Once the ML problem has been formulated, data wrangling (gathering, assessing, and cleaning) is done so that cleaned data is fed as input to the ML model. Training and testing of this ML model comprise the next steps. Python based libraries *pandas* and *scikit-learn* [9], [10] are used to implement the end-to-end ML workflow starting from data wrangling to model training and prediction. The ML workflow for the proposed solution followed in this paper is illustrated in **Error! Reference source not found.**. Once the ML model is finalized, it is integrated into an automated debug workflow comprising the steps shown in Figure 2. The three steps of debug automation shown in this figure are implemented through a python script. Each step of the solution is described in detail in the following subsections.

B. *Dataset description*

The data considered in this paper is based on old regression failure records. Each record has failure details like description, component, test name, context etc. The text in the description field of a record is parsed to extract any error message(s). The context field in the record has information about the root cause of the bug once it is debugged. The *bug-type* is identified as *design* or *non-design* based on this field. Example raw error messages extracted from the description field for each *bug-type* are shown as follows. Example *design* error message:*error,mod1 buffer in mod2inbuffarb functional block is full butstill getting write request primarily missingimplementationof fused mod3 support in mod4.Error: "/disk1/repo1/src/libs/mod5/mod5_buf.vs", 90: tb_top.top.mod6_evenbuf.full: at time 143201 psFiling bug to track this issue.Error: "/disk1/repo1/src/libs/mod5/mod5_buf.vs", 90: tb_top.top.mod6_evenbuf.full: at time 143201 psFiling bug to track this issue.* Example *non-design* error message:*Error:/disk2/repo2/src/units/mod7/mod7.vs",165:subsystem_tb.top.mod8.read_write_collision:Email thread attached:*

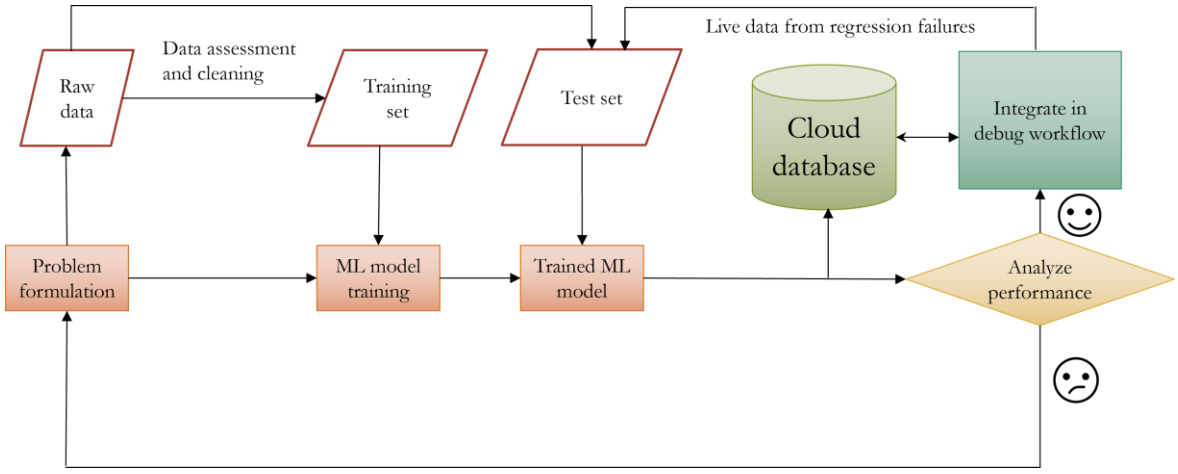

Figure 1. Machine Learning workflow for the proposed solution

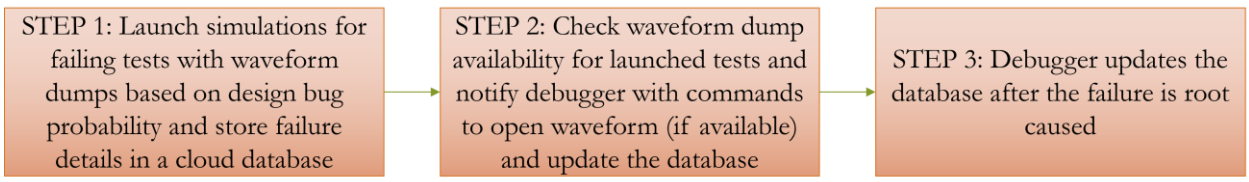

Figure 2. Automated debug workflow for the proposed solution

C. Data wrangling

Data collection: A Python based Application Programming Interface (API) based on SQL queries [11] is used to extract failure data of past records. This API returns the *description* field of a record in HTML format. The Python library *BeautifulSoup* [12] is used to extract all text data from the HTML output after some cleaning. The raw text data gathered is assessed and cleaned as follows.

Data assessment and cleaning: The collected raw data is cleaned extensively to reduce noise and keep only meaningful error messages. For example, the error message matching the regular expressions: (**Error.*@, ERROR.*@, Error.*:, ERROR.*:, ERROR\.*ns, \[ERROR\])** is used to filter the records as part of the cleaning process. The raw text data is also passed through some preprocessing steps commonly used in Natural Language Processing (NLP) like removing punctuation marks, removing email addresses, removing URLs etc.

D. ML model exploration and hyperparameter tuning

The problem being considered here is a binary classification problem with text data as input. Most of the popular ML models used for classification work on numerical data. So, the input text data needs to be converted to numerical data so that standard ML models can be used. The function *TfidfVectorizer* [14] available in *scikit-learn* is used to convert the text data into a matrix of numerical feature vectors. For example, if our input set consists of the two error messages mentioned in Section II.B, the corresponding matrix is of size 2x50 with fifty features. An ML model is trained with this set of numerical feature vectors as inputs and the corresponding *bug-type* (*design* or *non-design)* as the output (which is binary with two possible classes, *design* or *non-design*). A new test feature vector (from live regression data) is fed as input to the trained ML model and the expected *bug-type* is predicted as the output. Several ML models are tried to obtain the best possible prediction performance. Here classification performance is measured using *precision* and *recall* metrics [15]. Predicting an actual RTL *design* bug as a *non-design* issue is more expensive from the business perspective. So, improvement of *recall* for the *design* class is given more importance when judging the ML model performance without much compromise for the *non-design* class. The selection of optimal hyperparameters and preliminary results are discussed in the next section.

E. ML model integration into an automated debug workflow

The optimized ML model is integrated into a debug workflow consisting of three steps (mentioned in Figure 2) which are described below.

1. An input regression directory is scanned to identify tests with a specific error pattern which is mostly deterministic. For each such test, the error details (error message, relevant RTL file, RTL module, time, etc.) are extracted. Each error message is fed to the trained ML model to calculate the probability of the failure being a design bug. All this information is recorded in a cloud database (called fDB). A customized *.do file is created based on error details. A simulation run of the failing test is relaunched to generate a waveform dump if the design bug probability is above a certain threshold. The last step is to store the waveform dump path and waveform availability status in the fDB.
2. Query the fDB to check the simulation completion and waveform generation status of all tests launched. If the simulation run log has environment errors which can be fixed without manual intervention (e.g., freeing up disk space), the simulation is relaunched. Otherwise, an email is sent to the simulation environment team to fix the error and the waveform status is updated accordingly. Once the waveform is ready, an email is sent to the relevant validator(s) with instructions to load the waveform and update the fDB with debug details.
3. This step is used by the validator to update the fDB with useful comments after the failure is root caused. This is also used by the simulation environment team to update the fDB after fixing environment errors encountered while launching the failing tests.

III. Results & Discussion

A. ML model performance

The ML models used to solve the binary text classification problem were Logistic Regression, Support Vector Machines, Decision Trees, Random Forest, and Adaptive Boosted Decision Trees. The optimal hyperparameters for each ML model were chosen as those which achieved a minimum threshold on *design* precision and recall and had the best *non-design* recall on the cross-validation set. Table I shows the best *non-design* recall for various classifiers. Figure 3 shows the *design* precision and recall of different classifiers on the test set. Although the RandomForestClassifier achieves the highest *design* recall of 0.94, it has a *non-design* recall of 0.28. In contrast, the AdaBoostClassifier achieves a *design* recall of 0.82 and a *non-design* recall of 0.50. The AdaBoostClassifier also achieves a higher *design* precision compared to the RandomForestClassifier. Thus, overall, the classifier using

Adaptive Boosted Decision Trees was found to perform the best according to the criteria mentioned in Section II.D. Table II shows the hyperparameters for the optimized ML model used in the scikit-learn implementation [16] and its performance on the test set. The *recall* for the *design* class is 82%. This is also the most important metric to optimize for the problem being addressed here, as already discussed in Section II.D.

TABLE I
ML MODEL EXPLORATION

ML Model	**Best non-design recall**
LogisticRegression	0.52
LinearSVC	0.53
DecisionTreeClassifier	0.53
RandomForestClassifier	0.28
AdaBoostClassifier	0.50

TABLE II
OPTIMIZED ML MODEL HYPERPARAMETERS AND PERFORMANCE ON THE TEST SET

Metric	**Value(s)**
Optimal hyperparameters for AdaBoostClassifier	base_estimator=DecisionTreeClassifier(max_depth=1, min_samples_leaf=2, min_samples_split=6), n_estimators=80, learning_rate=0.3, algorithm='SAMME.R'
Precision for *design* class	0.59
Recall for *design* class (most relevant metric for the problem)	0.82
Precision for *non-design* class	0.76
Recall for *non-design* class	0.50

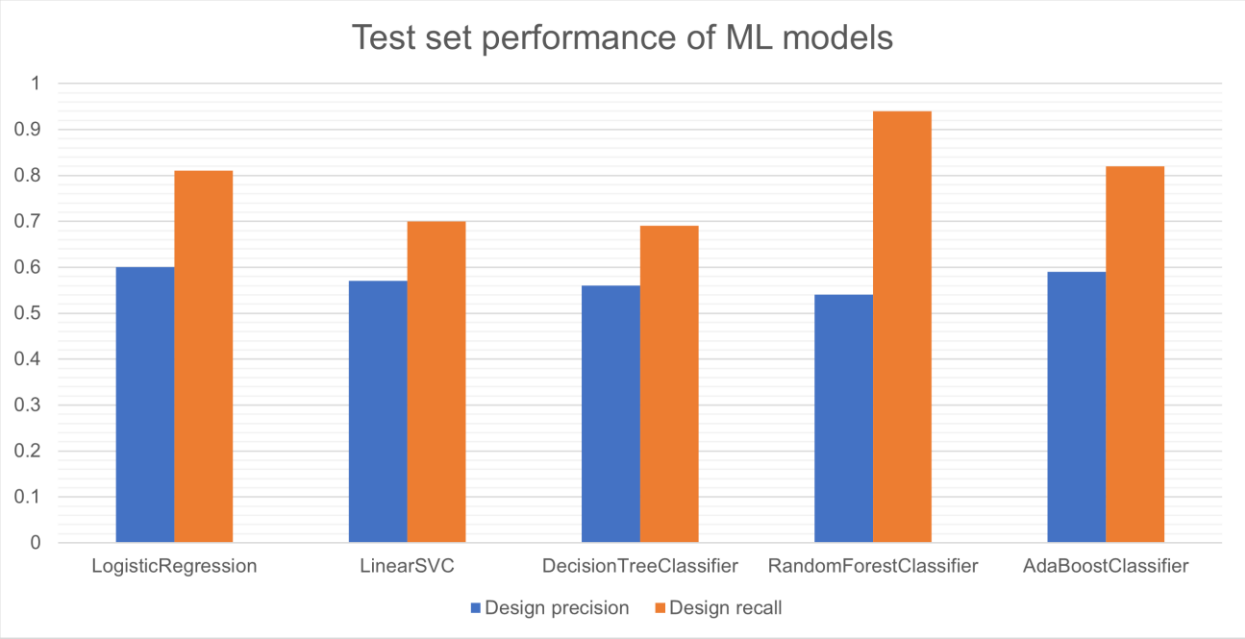

Figure 3. Classification performance of different ML models

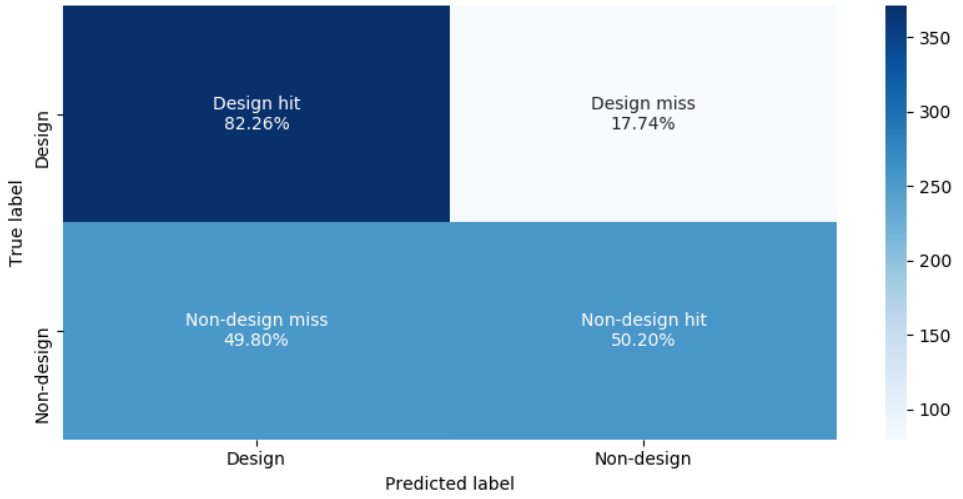

Figure 4. Confusion matrix for the test set with best ML model

Error! Reference source not found.4 shows the confusion matrix [17] based on predictions for the test set. The values in each of the four quadrants show the corresponding percentage of the true label in that row. The classifier fails to predict true *design* bugs correctly 17.74% of the time. The miss rate for true *non-design* bugs is higher, but not as expensive as missing an actual *design* bug.

B. ML model integration into debug automation workflow

The ML model code predicting design bug probability was integrated into the debug automation script with the three steps mentioned in Section II.E. A sample regression list of failing tests was chosen to run through the whole debug automation workflow. The database updates made through the script were manually observed and verified by accessing the cloud database using MySQL Workbench [18] on Windows. Figure 5 shows a sample snapshot of the updated database. The *Design_bug_probability* column serves as a useful tool to prioritize debugging failures which are likely pointing to RTL issues.

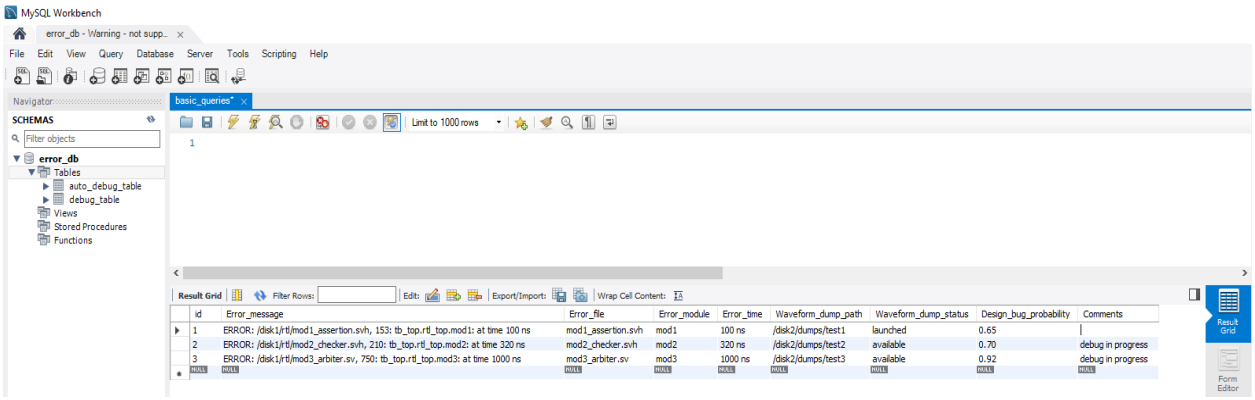

Figure 5. Sample cloud database snapshot after updates through the debug automation script

C. Implementation challenges

Most of the challenges faced in the implementation of the proposed solution were related to data wrangling. Extracting useful text from past failure records and obtaining cleaner error messages took a significant amount of time and effort. Finally, ML model training, testing and selection to optimize performance had to be done carefully based on the most relevant metric for classification from an application point of view.

IV. Conclusion and Future work

This paper explores a solution to the problem of predicting a hardware design bug using ML and aiding the debug process through automation. Although the proposed solution cares more about successful design bug prediction, the prediction of non-design bugs can also be improved significantly. As part of future work, the signal to noise ratio in the cleaned dataset can be investigated further, along with more diagnostics on the ML model performance. The improvement for precision and recall metrics can be further explored for both design and non-design bug predictions by increasing the dataset size, using more input features, exploring more powerful ML models etc. This framework can be further extended for more complex failures and prediction of other parameters relevant for successful design tape-out. There is also ample scope for exploration of more opportunities in all levels of validation where a similar ML approach can be applied.

References

[1] H. D. Foster, "Trends in functional verification: A 2014 industry study," in 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), June 2015, pp. 1–6.

[2] H. Foster. 2018 Wilson Research Group Functional Verification Study, Where Verification Engineers Spend Their Time", https://blogs.mentor.com/verificationhorizons/blog/2019/01/29/part-8-the-2018-wilson-researchgroup-functional-verification-study.

[3] S. Vasudevan," Still a Fight to Get It Right: Verification in the Era of Machine Learning," 2017 IEEE International Conference on Rebooting Computing (ICRC), Washington, DC, pp. 1-8, 2017.

[4] S. N. A. Saharudin, K. T. Wei, K. S. Na, "Machine Learning Techniques for Software Bug Prediction: A Systematic Review," Journal of Computer Science, vol. 16, p. 12, 2020

[5] J. Adler, R. Berryhill, and A. Veneris, "An extensible perceptron framework for revision rtl debug automation," in Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific, pp. 257–262, IEEE, 2017.

[6] S. M. Ambalakkat, E. Nelson," Simulation Runtime Optimization of Constrained Random Verification using Machine Learning Algorithms", Design and Verification Conf. (DVCON), 2019.

[7] S. Gogri, J. Hu, A. Tyagi, M. Quinn, S. Ramachandran, F. Batool, and A. Jagadeesh, "Machine Learning-Guided Stimulus Generation for Functional Verification" Design and Verification Conf. (DVCON), 2020.

[8] H. Jang, S. Yim, S. Choi, S. B. Choi "Machine Learning Based Verification Planning Methodology Using Design and Verification Data" Design and Verification Conf. (DVCON), 2022.

[9] https://pandas.pydata.org/

[10] https://scikit-learn.org/stable/

[11] https://en.wikipedia.org/wiki/SQL

[12] https://www.crummy.com/software/BeautifulSoup/

[13] Aurlien Gron. 2017. "Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems" (1st. ed.). O'Reilly Media, Inc.

[14] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

[15] https://en.wikipedia.org/wiki/Precision_and_recall

[16] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

[17] https://medium.com/@dtuk81/confusion-matrix-visualization-fc31e3f30fea

[18] https://www.mysql.com/products/workbench/